Overload Set Types

Document #:	P3312R1
Date:	2025-04-16
Project:	Programming Language C++
Audience:	EWGI

Reply-to:	Bengt Gustafsson <bengt.gustafsson@beamways.com>

Contents

1 Revision history

1.1 P3312R1

In R0 two different syntaxes were presented, where enclosing the function name in back-ticks was a way to a) ensure that a overload-set-type was created even if only one overload was visible, b) being able to use operator symbols in back-ticks to create an overload set containing both member and namespace scoped operators. In this version a) is left for solving in a future proposal and b) is removed and the classes we already have for each operator (std::plus, std::multiplies etc) will have to be used also in the future..

A history section was also added describing [P0119R2] and its rebuttal in [P0382R0]. Clarified that this proposal does not define the overload set as a generic lambda so it doesn't cause behavior change as P0119 did. Noted that if ADL behavior is needed a lambda can be used just like today.

1.2 P3312R0

Initial version presented in St Louis in June 2024.

2 Abstract

This proposal defines a type for each overload set of more than one function. Such *overload-set-types* are created when a placeholder type is deduced from the name of an overloaded function. In contrast with function pointers these types have no runtime state. This proposal does not specify any new keywords or operators, it just expands what placeholder types can be deduced from.

An object of *overload-set-type* can be called like the function overloads it represents, and overload resolution works exactly the same as if the overloaded function is called directly at the *point of deduction*. Additionally, an object of *overload-set-type* can be implicitly converted to the function pointer type of any of the overloaded functions it represents.

```
auto callWithFloat(auto&& f)
{
    double (*dfp)(double) = f;   // The appropriate overload of f is selected, error if none.
    return f(3.14f);    // If f is of overload-set-type overload resolution occurs here.
}

// As std::sin is overloaded the type of f above is an overload set type for std::sin at this
// point of deduction.
float x = callWithFloat(std::sin);
```

As this feature only relies on compile time overload resolution it works also for member functions, constructors, destructors and operators. For member functions and destructors the member function pointer call syntax must be used, while for constructors regular function call syntax is used. In this revision it is not possible to create an *overload-set-type* that contains both free function and member function operators, and depending on how the *overload-set-type* is constructed it has to be called with function pointer or member function pointer call syntax.

ADL does not affect the contents of *overload-set-types*. As there are no arguments with dependent types present at the point of deduction only overloads lexically visible at this point are included.

Each *point of deduction* may create a new *overload-set-type* just like how each lambda is of different type even if the source code is the same. When an *overload-set-type* is provided as a template argument at a point of template instantiation the overload set is formed at this lexical position.

As class and variable templates have only one point of instantiation only the overload set contents at that point is considered. This is consistent with calling the overloaded function without dependent argument types from within a member function of the class template or the initializer for a variable template.

For function templates each function invocation is a *point of instantiation* which may create a different instance if the argument types are different. Each time the same function name is provided as template argument to the same function template this is a new *point of deduction*. This may affect overload resolution within the instantiated function template specialization, so each call site is in principle instantiate separate specializations.

3 Motivating examples

Today you can't use functions that are overloaded when a callable's type is deduced. This is troublesome for instance with algorithms such as `std::transform` which often require wrapping a function in a lambda just because it is overloaded. With this proposal any function, overloaded or not, can be used in such scenarios.

3.1 Natural predicate syntax

Here are some examples involving `std::transform`.

```
std::vector<float> in = getInputValues();

std::vector<float> out;
std::transform(in.begin(), in.end(), std::back_inserter(out), std::sin);

// Or with ranges
auto out_r = std::views::all(in) |
             std::views::transform(std::sin) |
             std::ranges::to<std::vector>();

// Also works with operators, either using the operatorsyntax or the Class::operator syntax.
auto neg_r = std::views::all(out_r) |
             std::views::transform(&operator-) |           // Only free function operators
             std::ranges::to<std::vector>();
auto neg_r = std::views::all(out_r) |
             std::views::transform(&Type::operator-) |     // Only member operators
             std::ranges::to<std::vector>();
```

3.2 Even more perfect forwarding

Another problem that this proposal solves is that function pointers don't work with *perfect forwarding* when passing an overloaded function. Here is an example involving `std::make_unique`.

```
class MyClass {
public:
    MyClass(float (*fp)(float));
};

auto ptr = std::make_unique<MyClass>(std::sin);     // Works with this proposal!
```

Today `make_unique` can't be called with `std::sin` as argument although `MyClass` can be constructed with it. This is as the `auto&&` placeholder type of make_unique can't be deduced from a an overloaded function name.

3.3 Helping contract condition testing

This proposal also solves the issue encountered in [P3183R1] where `declcall` of [P2825R5] and macros had to combined to allow overloaded functions to be tested by its `check_preconditions` and `check_postconditions` functions. This proposal also allows operator, constructor and destructor contracts to be tested without further compiler magic.

```
// With P3183
#define CHECK_PRECONDITION_VIOLATION(F, ...) \
CHECK(!check_preconditions<__builtin_calltarget(F(__VA_ARGS__))>(__VA_ARGS__))

CHECK_PRECONDITION_VIOLATION(myOverloadedFunction, 1, "Hello");

// With this proposal:
CHECK(!check_preconditions<myOverloadedFunction>(1, "Hello"));

// And also
CHECK(!check_preconditions<MyClass::func>(MyClass{}, 1, "Hello"));
CHECK(!check_preconditions<operator+>(MyClass{}, 1));
CHECK(!check_preconditions<MyClass::MyClass>(1, "Hello"));
CHECK(!check_preconditions<MyClass::~MyClass>());
```

While it now seems unlikely that P3183 will move forward in its present form a similar facility is probably going to be defined and will need this feature.

4 History

Andrew Sutton presented the same basic "no syntax" idea in [P0119R2] but a flaw presented in [P0382R0] caused the no-syntax approach according to P0119 to change behavior in the case of forwarding a function name from a template function to another template function which could then select any overload present at the point of instantiation of the calling function template. If exactly one overload of the function name is visible where the function name is forwarded this works today, always calling the named non-overloaded function. If further overloads are introduced after the function template definition but before it is instantiated P0119 could cause breakage as such interveaning overloads could be selected.

This proposal circumvents this code breakage by not using a rewrite rule to a lambda containing an unqualified call to the named function. This is what caused the backwards compatibility issue in P0119 as ADL is used when the lambda is called, finding functions that would not be found in current C++. If an overloaded function name is used in template code in this proposal it can only select among overloads visible at the lexical location of the template definition in which the function name is mentioned.

5 Proposal

This proposal allows placeholder types to be deduced from names of overloaded functions. The proposal has no effect for functions that are not overloaded, these deduce the placeholder type to the function's function pointer type as today.

The *basic principle* of this proposal is that all aspects of the original function overload set are preserved when applying the function call operator to an object of *overload-set-type*. The different design decisions described below mostly stem from this basic principle.

5.1 Function categories

This proposal works for regular functions, member functions, constructors, destructors and to some extent for operators.

5.1.1 Regular functions

When a placeholder type is deduced from the name of an overloaded function an *overload-set-type* is synthesized and the function name is replaced by an instance of this type. For exposition purposes this type can be viewed as having a call operator for each function overload and a conversion operator to each function reference type. Each *overload-set-type* is regular and all instances compare equal. All overload resolution is done at compile time, only depending on the point of deduction of the *overload-set-type*. In contrast with a function pointer there is no runtime data to carry around, so the actual function calls compiled into the object code doesn't do runtime dispatch (except virtual dispatch if applicable).

Here is an example of a conceptual *overload-set-type* assuming that `std::sin` has two overloads:

```
// Exposition only
struct __std_sin_overload_set_type_1 {
    float operator()(float x) { return std::sin(x); }
    double operator()(double x) { return std::sin(x); }

    using F = float(float);
    using D = double(double);
    operator F&() const { return std::sin; }
    operator D&() const { return std::sin; }
};
```

This type is so conceptual that compilers are required to elide the call operator and function pointer conversion operators. The *overload-set-type* is just a way to express that the compiler remembers the contents of the overload set at the point of deduction. This means that the conversion operators above don't actually count as user defined conversions in overload resolution.

Although each *overload-set-type* is unnamed (just like a lambda) it can be retrieved using `decltype` (just like a lambda). As there are no data members an *overload-set* is always default constructible, copyable and assignable, but only within each *overload-set-type*, which means that you can't change the contents of of a variable of *overload-set-type* by assignment.

The function call operator can be applied to an object of *overload-set-type* just as if the original function was called. Overload resolutions also happens when a *overload-set*-type object is used inside a `declcall` construct of [P2825R5] while `static_cast` and binding to a function pointer or reference just selects one overload with matching signature if one exists.

Here are some examples of code valid with this proposal:

```
void compose(auto F, auto G, auto value) { return F(G(value)); }

double one = compose(std::tan, std::atan, 1);

auto s = std::sin;

using SinOverloads = decltype(s);

callWithFloat(SinOverloads());

void cc(float (*f)(float));

cc(s);
```

```
auto sptr = declcall(s(2.0f));

cc(sptr);

auto sptr2 = static_cast<float(*)(float)>(s);

cc(sptr2);

double x = s(3.14);

auto(std::sin);

auto{std::sin};

decltype(std::sin);

template<auto F> void myFun()
    requires requires { F(1.2); }
{
    F(2.3);
}

myFun(std::sin);
```

5.1.2 Member functions

The proposed feature also works when a placeholder type is deduced from an overloaded member function name prefixed with `&`. In this case the `(ref.*f)(args...)` or `(ptr->*f)(args...)` syntax must be used when calling the function for consistency with the case that the member function is not overloaded.

Overload resolution works exactly as if the overloaded member function was called directly. When an *overload-set-type* is deduced from a member function name it can always be used with an object reference or pointer, even if it contains `static` overloads and/or member functions with *explicit object reference*. This ensures that overload resolution works the same as if the member function was called directly. Note however that this differs from unoverloaded static functions and member functions with *explicit object reference* where the function pointer type is a non-member function pointer type and can only be called using function call syntax.

For overload sets that contain static member functions it is also possible to use regular function call syntax. This allows passing *overload-set-types* containing static member functions where unoverloaded static member function pointers can be passed today. Just as when using the `Class::function(args...)` syntax this fails if a non-static member function is selected by overload resolution. Note that calling using the regular function call syntax does not apply to member functions with *explicit object reference* as these can't be called like static member functions.

Converting an object of *overload-set-type* to a member function pointer type works the same as when converting a member function name directly. So if the function pointer type is for a free function there must be a matching *static* member function or member function with *explicit object reference* in the overload set.

```
struct B {
    virtual int f(int) { return 1; }
    static int f(float) { return 3; }
    int f(this B& self, double) { return 4; }
};
```

```
struct C : public B {
    int f(int) override { return 2; }
};

void callf(auto memp)
{
    B* bp = new C;

    (bp->*memp)(1);     // returns 2: Virtual dispatch happens.
    (bp->*memp)(1.2f);  // returns 3
    (bp->*memp)(2.3);   // returns 4

    int (B::*ip)(int) = memp;     // Set to overload for int
    int (*fp)(float) = memp;      // Set to static overload for float
    int (*dp)(B&, double) = memp; // Set to overload with explicit object reference
}

int main()
{
    callf(&B::f);
}
```

5.1.3 Constructors

An object of *overload-set-type* can be created by deducing a placeholder type from a constructor name of the form `&Class::Class`. Objects of type `Class` can subsequently be created by applying the function call operator to this object. As constructors don't have function pointer types even non-overloaded constructors deduce to *overload-set-types*. There is no possibility to convert an *overload-set-type* deduced from a constructor to a "constructor pointer" as there is no such thing.

In correspondence with the *basic principle* implicitly generated default, copy and move constructors are part of the *overload-set-type*.

5.1.4 Destructors

Even though destructors are not overloadable *overload-set-types* can be deduced from destructor names of the form `&Class::~Class` as there is no corresponding function pointer type. Objects of the deduced `overload-set-type` contain one overload which can be called just like a member function pointer to a parameterless function, but which can't be converted to a "destructor pointer" as there is no such thing.

5.1.5 Operators

An *overload-set-type* can be created by deducing a placeholder type from an operator in the form `operator@` for each *overloadable-operator* `@`. When an *overload-set-type* is deduced from an overloaded operator it follows the rules of calling operators using the `operator@(args...)` syntax. This means that in this proposal there is no way to create an overload set containing both free function and member function operators corresponding to when an operator is used via its *operator-token*. In keeping with the *basic principle* the built in operator overloads for fundamental, array and pointer types are included in operator overload sets when deduced from just `operator@`. For the same reason *overload-set-types* deduced from comparison operators include the synthesized operators created from opposite or argument swapped operators, as well as from the `<=>` operator.

Note that as we already have standard library types such as `std::less` for each operator the need to deduce an *overload-set-type* from an operator is less pronounced than for named functions.

An extension that enables creating *overload-set-types* containing both free function and member function operator overloads is a future possibility.

There is a rather theoretical problem with namespace scope operators: If there is exactly one user defined `operator@` on the namespace level visible when an *overload-set-type* is deduced it today deduces successfully to a member function pointer type. Thus when used it only has this one overload to select from, not the operators for built in types that were intended here. This proposal favors backwards compatibility and thus the operator name still deduces to a function pointer in this case. In reality there are usually a plethora of overloads visible in any namespace, so such operator function pointers can rarely be created today. Here is an example:

```
namespace test {
    class Foo {};

    Foo operator+(const Foo&, const Foo&) { return Foo(); }

    Foo (*plus_ptr)(const Foo&, const Foo&) = operator+;

    auto pp = operator+  // Works today: There is only one user-defined operator+ here.

    int a = 1 + 2;       // But ints can of course still be added!

    pp(1, 2);            // Fails today, pp is just a function pointer

    Foo operator+(const Foo&, int) { return Foo(); }

    auto pp2 = operator+ // This works with this proposal, and includes + for ints.

    pp2(1, 2);           // Works with this proposal
}
```

5.1.6 Conversion functions

In this proposal user-defined conversion functions are not included as there is no way to spell "all the conversion functions" and if a specific overload is named, for instance `&MyClass::operator int` this results in a member function pointer, so no *overload-set-type* can be deduced. A possible extension to handle this situation is described below.

5.2 Defaulted parameters

According to the *basic principle* of this proposal defaulted parameters are considered in the overload resolution process. By the same basic principle defaulted parameters are *not* considered when converting an object of *overload-set-type* to a function pointer.

5.3 Function Templates

If the overload set contains function templates these are included in the *overload-set-type* and selected by overload resolution as usual. If an explicit specialization is encountered after the point of deduction of the *overload-set-type* and gets selected by the overload resolution of a subsequent application of the function call operator to an object of the *overload-set-type* the program is ill-formed. Most compilers will warn in similar situations today and can continue to do so for this particular case.

5.4 Specifiers such as noexcept and constexpr

In keeping with the *basic principle* all specifiers on functions in the overload set are carried over to the *overload-set-type* and work the same as if the function name was used directly.

5.5 Contracts

Contracts on functions are evaluated in the same way when a function is called via an overload set type object as when the function is called directly.

5.6 No ADL

*Overload-set-type*s for unqualified free function names don't include overloads found by ADL. This is in contrast with generic lambdas containing function calls to overloaded functions using dependent argument types. Deducing a placeholder type from a namespace-qualified function name results in an *overload-set-type* containing only the overloads visible at this lexical position.

This rule ensures backwards compatibility to generic code that uses an (unoverloaded) function name as argument to a nested function call.

Example:

```
int Transformer(int);                                   // Unoverloaded predicate

template<typename Iter> void useTransformer(Iter begin, Iter end)
{
    std::transform(begin, end, begin, &Transformer); // Valid code today.
}

float Transformer(float);                               // Introduce another overload.

int main()
{
    std::vector<float> floats { 1.1f, 2.2f, 3.3f };
    useTransformer(floats.begin(), floats.end());    // float(float) overload not used.
}
```

5.7 Allowing std::invoke with all *overload-set-types*

`std::invoke` and similar functions in the standard library must work as expected for *overload-set-types* created from member function overload sets as well as for free function overload sets. If the invoke overload selection is made based on constraints rather than on matching function pointer or member function pointer signatures this should just work. As constraints are relatively new it is however likely that many `std::invoke` implementations will have to be rewritten to allow calling with an object of *overload-set-type* as the first argument. There should be no ABI breakage caused by this type of change as the selected overload of any `std::invoke` call that can be made today will retain the same signature.

In the fringe case that the overload-set-type contains both a non-static member function and a static member function that can be called using the same parameter list (i.e. the first parameter of the static function is a cvref qualified version of the containing class itself) an ambiguous call should result. This can easily be detected with a static_assert using requires expressions for the two call forms. Performing overload resolution to select which overload to call does not seem feasible to implement.

5.8 Rules for template instantiations

There are two conflicting requirements on the identity of *overload-set-types*. For understandability class/variable templates should work differently from function templates.

5.8.1 Class and variable templates

For class and variable templates we only want one specialization for each fully qualified function name. This is consistent with the fact that each class and variable template has just one point of instantiation, which is just

before the first declaration that needs the specialization . This ensures that static data members are instantiated only once and that virtual functions are unambiguously defined. As the fully qualified function name used to deduce the template parameter can be part of the mangled name of the template specialization its static data members, vtables and virtual functions can be shared between TUs (for variable templates this ensures that there is only one exemplar of each instance).

```
template<auto F> class Caller {
    void call() { F(1); }
};

void f(const char*);
void f(float);

Caller<f> callf;    // #1

void f(int);

callf.call();       // #2: Calls f(float)
Caller<f> call_f2;  // #3 No new instantiation of Caller.
callf2.call();      // Calls f(float)
```

At #1 the Caller class template is specialized for an *overload-set-type* containing two overloads of f. At #2, after a third overload of f has been declared, `Caller<f>::call` is instantiated, but as F is a template parameter of the class the third overload is not considered and `f(float)` is called.

Furthermore, at #3 there is no new class template instantiation as we already have a specialization of Caller for f.

For class and variable templates it is still the programmer that has to guarantee that the overload set contents is the same at the point of instantiation of each class template instance in all TUs linked together. Violating this is and ODR violation which is ill-formed no diagnostic required. If #3 was placed in a different TU than #1 such a violation would result and the linker may pick any of these two `Caller<f>::call` implementations. Note that this is exactly the same problem that would arise today if `Caller<F>::call` contained a call to `f` directly.

5.8.1.1 Out of line member function definitions

When directly calling an overloaded function from within a member function definition today the overloads visible in the lexical position of this definition are considered, not only the overloads visible at the class definition. This is not applicable in the overload-set-type case as the *point of deduction* is at the instantiation of the class template and even if a member function calls this overload-set-type object this does not affect where the point of deduction is. Here is an example:

```
template<auto F> class Caller {
    void call();
};

void f(const char*);
void f(float);

Caller<f> callf;                        // #1

void f(int);

template<auto F> void Caller<F>::call()  // #2
{
    F(1);
```

```
}

callf.call();                              // #3: Calls f(float)
```

Here the contents of the overload set is deduced at #1, so even if there is another overload of f visible at the definition of `Caller::call` at #2 this overload is not considered. This is true even as `Caller<f>::call` itself is not instantiated until #3.

Note that if f was called directly instead of via F the implementation at #2 would actually select the f(int) overload despite it being declared after the class template definition, while if `call` was defined inside the class definition, as in the previous example, f(float) would be called, despite the fact that `Caller::call` is not instantiated until its first use at #3 in this case too.

5.8.2 Function templates

For function templates it would be strange to only consider function overloads that were present at the first instantiation of the function template. It would also introduce scary long distance dependencies between call sites. Instead the user expectation is that there is a separate function template specialization for each call site even if the function name mentioned is the same. This follows from the fact that each time a placeholder type is deduced from an overloaded function name it is a unique type. Note that in the case of member functions of class templates using template parameters of the class template that were originally deduced from a function name the overload set is always the one deduced at the instantiation of the class template.

Here are some examples. Note that the contents of the overload set is *always* the overloads visible when the function name is mentioned.

```
template<auto G> callG()
{
    G(1);
}

void f(const char*);
void f(float);

auto FF = f;

callG<f>();     // #1 Calls f(float)

void f(int);

callG<f>();     // #2: Calls f(int).
callG<FF>();    // #3: Calls f(float)
```

Here the function template CallG gets instantiated twice for the same function name f, which makes the call at #1 selecting f(float) while the call at #2 selects f(int). Although the call of `callG<FF>` occurs after the declaration of f(int) the point of deduction is where FF was formed, so f(int) is not called.

The overload-set-type is different for each mention of the same function name even if the contents of the overload set happens to be the same the compiler basically has to instantiate the function template for each call site.

There are different ways for compilers to implement this, with different trade offs between compiler complexity, compile time and code size. To allow as many implementation strategies as possible the guarantee that the address of equal instantiations compare equal is dropped, similarly to the rules for inline function addresses.

One simple strategy is to never reuse function template specializations if any template argument is an *overload-set-type*. This idea is similar to viewing each deduction of a placeholder type as creating a new lambda. This

works but is taxing on compile time and object code size as many equal function specializations are likely to be generated.

A more elaborate strategy is to keep track of the overload set contents for a function name and somehow compare the current overload set contents to previously done instantiations and reuse the existing specializations if possible. Linkage is internal so there is no need for elaborate name mangling schemes, but code size and compile times within each TU are still reduced substantially.

To avoid code bloat caused by equal implemenations generated in several TUs it would be possible to involve the linker and allow it to do *comdat folding* of functions marked as being instantiations of the same function template for the same function name(s). The linker can then compare the object code for the functions to check which ones that can be merged into one.

It is also possible to actually use name mangling or some other metadata scheme to keep track of different instantiations between TUs to be able to share them without special handling by the linker. This alternative may result in very long mangled names (for instance for `std::swap`) but for functions with just a few overloads this may be feasible, so it would be possible for an ABI to define that up to N functions are handled by mangling, while object code for functions templated for larger overload sets is never shared. As modules become more prevalent metadata in the module files will have an easier time keeping track of which instances are equal.

As it is only differences in which functions that are actually selected by overload resolution for each call site inside the function template instance that affects its implementation it would also be possible to record which overload was selected at each call site involving the template parameter that was deduced to an *overload-set-type*. This brings the amount of information down from a potentially vast overload set to one function signature per call site in the function template instantiation. This is usually only one but it can be a few. With this strategy it is feasible to mangle all the information required to uniquely identify the template function instance so that its implementation can be shared between TUs without risking extremely long mangled names. However, this is novel in the respect that the mangled name can't be deduced from the function declaration, only from the definition, which can affect compile times as overload resolution for all call sites using an *overload-set-type* must be done before the compiler can know that the code has already been generated.

5.8.3 Virtual member functions

Virtual member functions have their point of instantiation at the point of instantiation of the containing class template. This is obvious as the class has only one point of instantiation and to create objects of the class its vtable is needed, which causes all virtual functions to be instantiated. Thus, if virtual functions use *overload-set-types* provided to the class template's template parameters the contents is defined from the point of instantiation of the class template. This is true also for out of line definitions of virtual member functions.

5.8.4 Member function templates

Member function templates can never be virtual and thus they can be instantiated according to the function template instantiation rules described above, leading to potentially multiple instantiations differing only by *overload-set-type* for member function template parameters.

5.8.5 Using class and variable templates inside function template definitions

When the template parameter type of a function template is explicitly used as a class or variable template template argument the class or variable template only has its point of instantiation if the function name of the *overload-set-type* has not been used for instantiation before. This ensures that there is only one class and variable template instance for each function name even in this situation.

In this example we try to count calls *to any overload* of a function by wrapping calls in a `countedCall` function template.

```
template<auto F> struct CountCalls {
    auto operator()(auto... args) { count++; return F(args...); }
    static inline size_t count = 0;
```

```
};

int f(int) { return 1; }

int countedCall(auto F, auto... args)
{
    CountCalls<F> counting;             // #1
    return counting(args...);
}

int a = countedCall(f, 1.2);            // #2 a is set to 1.

int f(double) { return 2; }

int b = countedCall(f, 1.2);            // #3 b is set to 2.

// This is not true, the first increment was to CountCalls<int(int)>::count
assert(CountCalls<f>::count == 2);

int f(const char*) { return 3; }

int c = countedCall<f>("Hej");          // #4 c is set to 3.

// This is true, there is only one CountCalls instance for overload set type of f.
assert(CountCalls<f>::count == 2);
```

This implementation does not work as intended as f is not overloaded when `countedCall` is called the first time at 1, so F is a function pointer type and thus there are two different `CountCalls` specializations, one for this function pointer type and one for the *overload-set-type* of f.

On the other hand, the call at #4, which instantiates `countedCall` with a new *overload-set-type* for f containing the int(const char*) overload this refers to the one and only specialization of `CountCalls<f>` that was instantiated by the instantiation of countedCall at #3.

The first revision of this paper solved this by using the backtick character to force a function name to result in an overload-set-type even if the function was not overloaded. This was a complicated solution to a very fringe problem and in this revision we just acknowledge that there is such a problem and punt any solution to a future proposal.

5.9 Explicit instantiation

A class or variable template can be explicitly instantiated as usual even with its template parameters deduced to *overload-set-types* and as usual the contents of the overload set at the first instantiation is used in the implementation. This also explicitly instantiates the non-template functions of class templates which is not problematic as there is only one contents of any *overload-set-types* their implementations can reach.

However, a function template should not be explicitly instantiated for overload-set-types as this would be misleading and does not allow the function definition to be hidden from any call sites, much like an inline function. An alternative is to allow but ignore explicit instantiations. The important part is that the definition must be visible when calls are made.

```
template<typename F> void f(F&& func);

extern template f(&std::sin);       // Not allowed with this proposal, see below.
```

Another way of thinking is that when an explicit function template instantiation is done the *intent* is to lock the

overload-set-type(s) affecting the instantiation so that further calls for the same function name are forced to use the explicit function instantiation even if the contents of those overload sets has changed. This is logical if the explicit instantiation is viewed as a way to tell the compiler which *point of instantiation* to use when generating code for the function.

Which way to resolve this is still an open question, but as explicit instantiation of function templates is not a widely used feature it may be good enough to just forbid explicit instantiation if an overload-set-type is deduced when doing so. This does not break old code as unoverloaded function pointer types can still be used.

5.10 No equivalence of overload set types

When the type of a function is deduced using `decltype` and this turns out to be an overload-set-type it is possible to create objects of this type. But allowing assignment between such objects may cause problems as what looks like the same type may not actually be. There are some possibilities, but let's start with an example.

```
int f(int);

using F1 = decltype(f);     // A function pointer type.

int f(double);

using F2 = decltype(f);     // An overload set type
using F2b = decltype(f);    // An overload set type

int f(const char*);

using F3 = decltype(f);     // Another overload set type.

F1 f1;
F2 f2 = f1;     // Illegal
F3 f3 = f2;     // Illegal
F2 f4 = f;      // Illegal
F3 f5 = f;      // Illegal: Point of deduction (for f here!) differs.
F2b f2b = f2;   // #1

F1 f6 = f1;     // #2
F2 f7 = f2;     // #3
```

Here we create three different types for f when it has 1, 2 and 3 overloads. We then try to initialize variables of these fixed types from variables of the other types. This can be treated in different ways. First observation is that regardless of how the value is initialized it has its own type related to the contents of the overload set of f at the point of deduction, i.e. the decltype call, so the only issue is in which cases there should be an error.

This proposal makes all these initializations illegal except the last two. At #2 we are just assigning function pointers which must continue to work. At #3 we are assigning between objects of the type of the same decltype use so it should be easy to allow.

The problematic case is #1 where where have two decltype-generated types that *happen* to have the same functions in the overload set. To allow this and make those types the same would require keeping track of the entire overload set.

As these initializations and the corresponding assignments have no effect anyway it is proposed to make it illegal to mix objects of *overload-set-type* when the type has initially been deduced separately, regardless of if the contents of the overload set has changed between these deductions or not. This simplifies compilers.

Note: The exception to this rule is that when class and variable templates are instantiated all overload-set-types for the same fully qualified function name are considered to be the same, with the contents of the overload fixed to the overload-set-type used when instantiating the class or variable template.

6 Possible extensions

None of these extensions are part of this proposal revision, but can be added to the language at a later time.

6.1 Allowing member function references

Today there is no such thing as a member function reference. Thus we for consistency require the qualified member function name to be explicitly converted to a member function pointer using a prefix `&` operator. This is what this proposal suggests.

Introducing member references could be a good idea, but it is another proposal. This has its own complications as the rules for automatic conversion from free functions to function references and function pointers are somewhat contrived due to historical reasons, and decisions have to be made as to if member pointers and references should work the same or deviate somehow.

6.2 Calling using regular function call syntax.

It would be possible to allow using regular function call syntax on objects with *overload-set-type* even if the type is deduced from an overloaded member function, member operator or destructor. Providing the implicit object reference as the first argument would be required at each call site, just as for `std::invoke`. If the overload set contains static member functions these would then be represented by two synthesized overloads in the *overload-set-type* as you can call a static function using the `obj.static_function()` syntax although `obj` is not used. This causes a risk for ambiguity not present when calling the static member function directly.

The advantage of this idea is that the user of the *overload-set-type* object does not have to know if the `overload-set-type` was deduced from a free function or a member function. This is a very limited form of *unified function call syntax* (UFCS) and does not solve the main use case for UFCS where for instance `begin` and `end` can either be free functions taking an object or a member of that object's type.

To make this functionality useful it would have to be made possible to call a *member function pointer* as a regular function, providing the implicit object reference explicitly. Otherwise we get functionality that *only* works if the member function is overloaded. On the flip side we today have a situation where static and explicit object reference (deducing this) functions work differently from non-static member functions in this respect, which this extension would unify.

```
struct MyClass {
    void f();
    static void g();
    void h(this MyClass& o);
};

auto fp = &MyClass::f;
auto gp = &MyClass::g;
auto hp = &MyClass::h;

MyClass o;

(o.*fp)();      // Ok
(o.*gp)();      // Error today
(o.*hp)();      // Error today
fp(o);          // Error today
gp(o);          // Error today
hp(o);          // ok

gp();           // ok
```

For consistency with the overloaded case as defined above all possibilities for calling in the example must be allowed, where the calls to g just ignore the object reference.

As extending rules for member function pointers in this way is a prerequisite for extending overload-set-type objects with free function callability this is not included in this proposal. Instead this would be an addition to a UFCS proposal to make sure it works consistently between named functions and *overload-set-type* objects.

6.3 Deduction from operator tokens

To avoid above mentioned shortcoming for operators, that there is no way to create an *overload-set-type* containing both free function and member function operator overloads it would be nice to be able to deduce an placeholder type from the *operator-token* itself. Then we could write code like this:

```
std::vector<float> in = getInputValues();

std::vector<float> out;
std::transform(in.begin(), in.end(), std::back_inserter(out), -); // Invert values

auto plusses = +;                       // All overloads of +
auto free_plusses = operator+;          // Only free function overloads of +
```

This functionality is the same for operators that can be defined both as free functions and member functions and for those that can only be defined as member functions.

To make this work a new production in the *assignment-expression* rule can be added. As this production only contains an *operator-token* there is no risk of parsing ambiguity. The only strain on the compiler is that if an expression starts with an *operator-token* the next token must be checked to see if is a comma, semicolon, right parenthesis, bracket or brace. If so select the *operator-token* only case whereas for any other token parsing proceeds as today, eventually consuming the *operator-token* in *unary-expression*.

assignment-expression:
conditional-expression
logical-or-expression assignment-operator initializer-clause
throw-expression
operator-token

It would be possible to forbid expressions consisting of only an *operator-token* when not used to deduce a placeholder type, but this seems complicated wording wise, and compilers usually have warnings like *statement has no effect*, in these cases which may be sufficient to filter out the case where a stray *operator-token* is mistakenly typed.

The reason for placing this production in the assignment-expression rule is to be able to use *overloaded-operators* as function arguments. If it was placed in the *expression* rule any usage as a function argument would have to be enclosed in parentheses. A third option is to *mandate* surrounding parentheses at all use by instead introducing a new production in *primary-expression* adding a third type of parenetheses there along with fold expressions and nested expressions. This approach may be safer and the extra parenthesis highlights that something special is going on.

```
std::vector<float> in = getInputValues();

std::vector<float> out;
std::transform(in.begin(), in.end(), std::back_inserter(out), (-)); // Invert values

auto plusses = (+);                     // All overloads of +
auto free_plusses = operator+;          // Only free function overloads of +
```

An expression of the form `(@)` will evaluate to an object of *overload-set-type* regardless of whether a placeholder type is deduced from it. This seems necessary as there is no other type this expression could have. This means

that by coincidence it is now possible to write `(@)(lhs, rhs)` or `(@)(arg)` due to how the grammar works.

An advantage of this formulation is that we can still never write two consecutive commas, whereas with the production in *assignment-expression* we can write `auto x = ,,1;` as the first comma is an *assignment-expression* but then the *expression* production including the comma operator is employed so the second comma causes the first comma to be ignored and x is initialized to the integer 1.

As an alternative another syntax which allows for function names can be used, this is described below, as an alternate extension.

6.3.1 Supporting fundamental, pointer and array types

When placeholder types are deduced from operators it is natural that "operator overloads" for fundamental types are included, as it improves the consistency to the direct use of the operator token, all according to the *basic principle*.

For pointers there are a few operators defined as well as the dereference operators. With this extension those operators should also be included in the overload set for the relevant operators.

For arrays it would be most likely that the pointer operators are used after *array-to-pointer* decay, but this doesn't matter semantically.

6.4 Supporting overloaded conversion functions

It may be possible to create an *overload-set-type* from all the conversion operators of a class, but this would require some specific new syntax such as `&MyClass::operator typename` to be able to denote this overload set. Overload set types of this variety would be very special just as overloaded conversion operators themselves, as the overload selection happens due to the *required type* rather than the argument types (which is always the object itself). This type of backwards overload resolution is already implemented in compilers so it doesn't seem overly complicated to allow this too, except for the special keyword parsing.

```
class MyClass {
    operator int() { return 0; }
    operator double() { return 1; }
};

auto conv = &MyClass::operator typename;

MyClass c;

int a = (c.*conv)();
double a = (c.*conv)();
```

This feature would not be needed for [P3183R1] to test contracts on conversion functions as each overload has its own name which enables for instance:

```
CHECK(!check_preconditions<MyClass::operator int>(myObject));
```

In fact it seems rather unnecessary to support overload sets of conversion operators given that it requires a syntactical extension and seems more risky when it comes to implementation effort depending on compiler internals. The only reason to support this seems to be completeness.

7 Implementation experience

None so far.

8 Acknowledgements

Thanks to Jonas Persson for valuable insights regarding uniqueness of the overload-set-types.

Thanks to Joe Gottman for feedback on P3183R0 which spurred adding constructor and destructor support here.

Thanks to my employer, ContextVision AB, for sponsoring my attendance at C++ standardization meetings.

9 References

[P0119R2] Andrew Sutton. 2016-05-28. Overload sets as function arguments.
https://wg21.link/p0119r2

[P0382R0] Tomasz Kamiński. 2016-05-29. Comments on P0119: Overload sets as function arguments.
https://wg21.link/p0382r0

[P2825R5] Gašper Ažman. 2025-03-17. Overload resolution hook: declcall(unevaluated-call-expression).
https://wg21.link/p2825r5

[P3183R1] Bengt Gustafsson. 2024-05-22. Contract testing support.
https://wg21.link/p3183r1